FPGA Design Deobfuscation by Iterative LUT Modification at Bitstream Level

Extended version

Michail Moraitis Elena Dubrova
Department of Electrical Engineering, Royal Institute of Technology (KTH)
Electrum 229, 196 40 Stockholm, Sweden
{micmor,dubrova}@kth.se

***Abstract*—Hardware obfuscation through redundancy addition is a well-known countermeasure against reverse engineering. For FPGA designs, such a technique can be implemented with a small overhead, however, its effectiveness is heavily dependent on the stealthiness of the redundant elements. Hardware opaque predicates can provide adequately stealthy constant values that can be used for obfuscation. However, in this report, we show that such obfuscation schemes can be defeated by ensuring the full controllability of each active look-up table input in a design via iterative bitstream modifications. We present an algorithm that works directly on the bitstream and does not require the possession of a netlist. The feasibility of our approach is verified with the example of an obfuscated SNOW 3G design implemented in a Xilinx 7-series FPGA.**

***Index Terms*—Obfuscation, hardware opaque predicate, SRAM FPGA, bitstream modification, reverse engineering.**

I. Introduction

With the growth in popularity of Field-Programmable Gate Arrays (FPGAs) for the implementation of cryptographic algorithms and Artificial Intelligence (AI) acceleration, a number of FPGA-specific security challenges arise. For instance, it has been demonstrated that bitstream modification attacks make it possible to recover the secret key from FPGA implementations of cryptographic algorithms [1]–[9]. In the future, when AI algorithms become a natural part of many systems, the extraction of neural network models from FPGA bitstreams by reverse engineering might also pose a serious threat. Thus, developing defense mechanisms against FPGA bitstream reverse engineering and bitstream modification, as well as testing their resilience against potential weaknesses, is critical.

A popular countermeasure against SRAM FPGA bitstream reverse engineering and bitstream modification is design obfuscation. Obfuscation attempts to transform a design into a functionally equivalent, but structurally different representation which is more difficult to understand. Obfuscation can be applied at different levels of abstraction [8], [10]–[14]. Typically obfuscation involves some kind of redundancy addition. Since there are powerful Automatic Test Pattern Generation (ATPG), Binary Decision Diagram (BDD), and Satisfiability checking (SAT)-based tools for combinational redundancy identification and removal, opting for sequential redundancy, such as the hardware opaque predicates [15], [16] is thought to result in stronger obfuscation.

This work was supported in part by the research grant 2021-02426 from Vinnova and by the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications at KTH Royal Institute of Technology.

A simple example of a hardware opaque predicate is an n-stage Linear Feedback Shift Register (LFSR) with an n-input OR gate taking inputs from all LFSR stages [15]. If the LFSR is initialized to any non-zero state, the output of the OR is always 1 since any subsequent LFSR state contains at least one 1 value. However, a drawback of this approach is that LFSRs can be found relatively easy [4]. More stealthy hardware opaque predicates based on counters and Finite State Machines (FSM) were presented in [16]. To the best of our knowledge, no methods for identifying such constructions are known at present, especially if a netlist is not available.

Our Contribution: In this paper, we demonstrate that, by assuring the full controllability of each input of each instantiated Look-Up Table (LUT) in an FPGA design via iterative LUT modification, we can defeat obfuscation based on hardware opaque predicates *regardless of how stealthy they are* (unless functional duplication is employed to mask the effect of undetectable faults they create). The key idea is to not search for the hardware opaque predicate itself, but rather for the LUT inputs that behave as undetectable stuck-at faults during the execution of the algorithm under attack. The presented method employs bitstream reverse engineering to determine the logic functions implemented by LUTs and the wires connected to the LUT inputs. This is an advantage over approaches that require a netlist. We demonstrate the feasibility of our approach on the example of an obfuscated SNOW 3G design implemented in a Xilinx 7-series FPGA.

Paper Organization: The rest of the paper is organized as follows. Section II presents background information necessary for understanding the paper. Section III reviews previous work. Section IV describes the proposed deobfuscation method. In Section V, the presented method is applied to an obfuscated SNOW 3G design. Section VI concludes the paper and discusses open problems.

II. Background

This section presents background information on FPGAs.

A. Bitstream representation of FPGA basic building blocks

An FPGA fabric is a mesh of Configurable Logic Blocks (CLBs) implementing user-defined logic that is connected through routing channels that pass through programmable switch boxes. By defining both, the functionality of the logic elements and their interconnections, a physical circuit is created on this mesh. In this subsection, we describe the basics of logic and routing in Xilinx 7 series FPGAs and their representation in the bitstream.

1) Look-Up Tables: In SRAM FPGAs, CLBs typically consist of k-input LUTs. In Xilinx 7 series FPGAs, $k = 6$, thus a LUT can implement a Boolean function of up to 6 variables. Regardless of the actual number of inputs the LUT's function depends on, its truth table appears in the bitstream as a 64-bit vector, called *initialization vector*, partitioned into four 16-bit words which are placed on a fixed distance from each other. Each bit of the truth table represents the output value of the LUT for a specific input assignment. If some of the LUT inputs are not used, they are treated as *don't cares* in the truth table and their value is by default fixed to constant-1 from the synthesis tool.

2) Programmable Interconnect Points: The routing in FPGAs is performed through Programmable Interconnect Points (PIPs). A PIP is a connection between two points: a *source* and a *destination PIP junction*.

Activating or deactivating a PIP in the bitstream results in creating or removing the connection between the corresponding source and destination PIP junctions. Each FPGA board has a predefined set of PIPs that dictates the possible ways that PIP junctions can be connected to each other. In other words, two PIP junctions cannot be connected unless there is a PIP that allows this connection.

In Xilinx 7 series FPGAs, there are two types of PIPs: PIPs that appear in the bitstream and PIPs that do not. Following the terminology of project Xray [17], we call PIPs that do not appear in the bitstream *fake* and PIPs that do appear *regular*.

Table I lists the format of the three PIP junction types from the project Xray database [17] which are relevant for this paper.

B. Architecture of Xilinx 7 series FPGA

The fabric of Xilinx 7 series FPGAs is a grid of *tiles* uniquely identified by their X and Y coordinates. There are different types of tiles. In the presented method, we use *interconnect tiles* (INT tiles) and *configurable logic block tiles* (CLB tiles).

The INT tiles are responsible for the majority of routing. An INT tile is a large switchbox consisting of a set of PIP junctions. A CLB tile has a small switchbox connected horizontally to an INT tile on one side and to two blocks called *slices* (which are also contained in the CLB tile) on the other side. If a CLB tile is on the right side of its corresponding INT tile, then they are both labeled as *right*. Otherwise, they are labeled as *left*.

TABLE I
PIP Junction Formats

PIP junction formats	
PJ1	$CLB < x1 > \_ < x2 > \_ < x3 >$
PJ2	$CLB < x1 > \_IMUX < x4 >$
PJ3	$IMUX\_ < x5 >< x4 >$
Parameter Values	
$x1$:	*CLB identifier*:
	LL for a CLB with two sliceLs
	M for a CLB with one sliceL and one sliceM
$x2$:	*Vertical Slice identifier*:
	L for an top slice
	LL for a bottom slice when $x1$ =***LL***
	M for a bottom slice when $x1$ =***LM***
$x3$:	*LUT input identifier in Slice*:
	$A1 \ldots A6$
	$B1 \ldots B6$
	$C1 \ldots C6$
	$D1 \ldots D6$
$x4$:	*LUT input identifier in CLB*:
	positive integer in range $[0, 47]$
$x5$:	*Positional CLB identifier*:
	null for right interconnect tiles
	L for left interconnect tiles

Each slice contains four LUTs, eight flip flops (FFs), a fast carry logic unit, and multiplexers (MUXes) to control the internal routing. The slices are positioned vertically inside a CLB, thus they are usually referred to as *top* and *bottom* slices. Slices are also categorized as *SliceM* or *SliceL* depending on whether they contain conventional LUTs (SliceL) or special LUTs that can be also configured into a 32-bit shift register or a distributed LUT-based RAM (SliceM).

III. Previous Work

This section describes previous work on FPGA design obfuscation and related attacks.

Logic locking is one of the most popular approaches for protecting intellectual property. It is based on embedding into a design a secret key that needs to be supplied for the design to function correctly [11]–[14]. A comprehensive overview of logic locking techniques can be found in [18]. In FPGA, the overhead related to key embedding can be minimized by employing unused LUT inputs, referred to as *dark silicon* [11]. The term *occupancy* describes the percentage of actually used LUT inputs in a design. The occupancy of FPGA designs can be quite low, e.g. an average occupancy of 30% is reported in [11] for nine benchmark designs. Therefore, finding unused LUT inputs to embed the key is typically not a problem.

In its essence, the utilization of unused LUT inputs is equivalent to injecting *undetectable stuck-at faults*, which do not cause incorrect output values for any input assignment

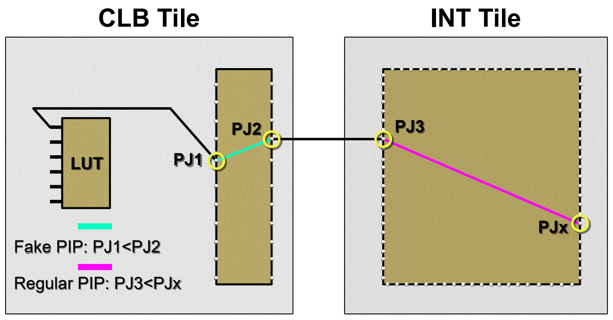

Fig. 1. Visualisation of PIPs connected to LUT inputs.

during the execution of the protected algorithm. A simple way to create an undetectable stuck-at fault is to set a net to constant-0 or constant-1 by connecting the net to GND or VCC, respectively, and changing the corresponding part of the LUT's truth table [8]. However, such constants are easy to find if the interconnect bitstream format is known [19]. A constant can also be created using a combinational logic circuit with redundancy, e.g. $x + \overline{x}$ for the constant-1 or $x \cdot \overline{x}$ for the constant-0, or a sequential circuit such as an LFSR [15], a counter, or an FSM [16]. In software obfuscation, a function providing a constant Boolean output regardless of the input is called *opaque predicate*, hence the name *hardware opaque predicate* used in [15], [16].

In combinational circuits, undetectable stuck-at faults can be identified using ATPG [20]–[22], SAT [23]–[29] and fault-independent methods [30]–[32]. ATPG and SAT algorithms can guarantee the detection of all undetectable stuck-at faults, but their worst-case time complexity is exponential. Fault-independent methods cannot always find all undetectable faults, but they have the advantage of polynomial worst-case time complexity. SAT-based attacks in particular have drawn a lot of attention, with many methodologies presented to resist them [33]–[36] and also many to enhance them [23]–[29].

Another type of redundancy that can be used for obfuscation is *functional duplication* which occurs when different sub-circuits implement the same function. The functional duplication in combinational circuits can be identified using SAT [37], BDD sweeping [38] and structural hashing [39]. Both, SAT and BDD sweeping, guarantee detection of all functional duplications, but they have an exponential worst-case time complexity. Structural hashing can identify structurally isomorphic equivalent sub-circuits in linear time. For this reason, obfuscation methods using functional duplication typically implement duplicated blocks in a diverse manner [13].

IV. Deobfuscation Algorithm

In this section we present the new deobfuscation algorithm FindObfuscated(). Its pseudo-code is shown as Algorithm 1.

FindObfuscated() takes as input a bitstream, $\mathcal{B}$, and returns a list of potential deobfuscated LUT candidates. Some of the candidates may be non-obfuscated LUTs in reality, i.e. false positives are possible.

Algorithm 1 An algorithm for finding obfuscated LUTs.

```
Name: FindObfuscated($\mathcal{B}$)
Input: Bitstream $\mathcal{B}$
Output: Set of deobfuscated LUT candidates
1:  $\mathcal{S} = \emptyset; \mathcal{M} = \emptyset$;
2:  $\mathcal{P}$ = PIP_Extract($\mathcal{B}$); /*$\mathcal{P} := ((p_{1,1},\dots,p_{1,k_1}),\dots,(p_{n,1},\dots,p_{n,k_n}))$, where $p_{i,j}$ is the PIP associated with the $j$th input of LUT $l_i$, $i \in \{1,\dots,n\}$, $j \in \{1,\dots,k\}$*/
3:  $\mathcal{L}$ = LUT_Extract($\mathcal{B}$); /*$\mathcal{L} := ((l_1,c_1),\dots,(l_n,c_n))$, where $c_i$ is the coordinate of LUT $l_i$ in $\mathcal{B}$, $i \in \{1,\dots,n\}$*/
4:  Clean($\mathcal{L},\mathcal{P}$)
5:  for each $i \in \{1,\dots,n\}$ do
6:    for each $j \in \{1,\dots,k_i\}$ do
7:      for each $\alpha \in \{0,1\}$ do
8:        $\mathcal{B}^* =$ Replace$(\mathcal{B}, l_i, c_i, j, \alpha)$; /*Replaces LUT $l_i$ at coordinate $c_i$ by a LUT $l_i^\alpha$ in which the $j$th input is stuck-at-$\alpha$*/
9:        Upload $\mathcal{B}^*$ to the FPGA
10:       if $\mathcal{B}^*$ generates the same output as $\mathcal{B}$ then
11:         if $(l_i^{\overline{\alpha}}, c_i, j, \overline{\alpha}) \notin \mathcal{S}$ then
12:           $\mathcal{S} = \mathcal{S} \cup (l_i^\alpha, c_i, j, \alpha)$;
13:         else
14:           $\mathcal{S} = \mathcal{S} - (l_i^{\overline{\alpha}}, c_i, j, \overline{\alpha})$;
15:         end if
16:       end if
17:     end for
18:   end for
19: end for
20: Count the number of occurrences of each LUT $l$ in $\mathcal{S}$, $N(l)$
21: for each $l$ in $\mathcal{S}$ such that $N(l) > 1$ do
22:   for each subset $J \subseteq \{j_1,\dots,j_{N(l)}\}$ of size $|J| > 1$ do
23:     $\mathcal{B}^* =$ Replace$(\mathcal{B}, l, c, J, \mathrm{A})$;
24:     Upload $\mathcal{B}^*$ to the FPGA
25:     if $\mathcal{B}^*$ generates the same output as $\mathcal{B}$ then
26:       $\mathcal{M} = \mathcal{M} \cup (l^{\mathrm{A}}, c, J, \mathrm{A})$;
27:     end if
28:   end for
29: end for
30: return $\mathcal{S} \cup \mathcal{M}$
```

First, a list of all active PIPs connected to instantiated LUT inputs is extracted from the bitstream. This list is represented by a vector $\mathcal{P} = ((p_{1,1},\dots,p_{1,k_1}),\dots,(p_{n,1},\dots,p_{n,k_n}))$, where $p_{i,j}$ is the PIP associated with jth input of the LUT l_i, for $i \in \{1,\dots,n\}$, $j \in \{1,\dots,k\}$.

Each input of a LUT is connected to a PIP junction of type $PJ1$ in the CLB's switchbox (see Fig. 1). This PIP junction forms a fake PIP with a PIP junction of type $PJ2$, which in turn is connected to a PIP junction of type $PJ3$ located in the corresponding INT tile switchbox. If the input is not used, then $PJ3$ forms a fake PIP with PIP junction VCC_WIRE (constant-1). If the input is used, then $PJ3$ forms a regular

PIP with one out of 25 possible PIP junctions in the INT tile switchbox. So, if the bitstream contains an activated PIP with the destination $PJ3$, the corresponding LUT input is used somewhere in the design.

Next, a list containing the truth tables of all instantiated LUTs with their coordinates in the bitstream is extracted. This list is represented by a vector $\mathcal{L} = ((l_1, c_1), \ldots, (l_n, c_n))$, where c_i is the coordinate of LUT l_i in $\mathcal{B}$, $i \in \{1, \ldots, n\}$.

At step 5, the procedure CLEAN is called with $\mathcal{P}$ and $\mathcal{L}$ as arguments to remove possible don't-cares in the LUT function truth table. Obfuscation techniques such as [8] use these don't-cares to camouflage LUT's truth table without adding any new input to the LUT and so does the watermarking scheme presented in [40]. Since there is a one-to-one mapping between LUT inputs and PIPs involving $PJ3$, the sub-vectors $(p_{i,1}, \ldots, p_{i,k_i})$ of $\mathcal{P}$ provide information about k_i input variables on which the function of the LUT l_i actually depends. Leveraging that, CLEAN updates the truth table of every LUT in $\mathcal{L}$ accordingly. Note that, in a non-obfuscated bitstream, this step would be unnecessary since this is how vendor tools format LUT truth tables by default.

At steps 6-20, for each LUT $l_i \in \mathcal{L}$ and each of its instantiated inputs $j \in \{1, \ldots, k_i\}$, the truth table of l_i in $\mathcal{B}$ is modified to a truth table in which the jth variable is stuck-at-α. This is done by replacing $f|_{x_j=\overline{\alpha}} = f|_{x_j=\alpha}$ where $f|_{x_j=\alpha}$ denotes a subfunction of the function $f(x_1, \ldots, x_k)$ of the LUT l_i in which $x_j = \alpha$ and $\overline{\alpha}$ is the Boolean complement (NOT) of α. The modification is done directly in the bitstream.

The resulting modified bitstream $\mathcal{B}^*$ is uploaded to the FPGA to compare its output sequence to the one of the original bitstream $\mathcal{B}$. If the sequences are the same and l_i with the jth input fixed to $\overline{\alpha}$ is not yet in the list of candidates, $\mathcal{S}$, then l_i is added to $\mathcal{S}$ along with its coordinate c_i, input j and stuck-at fault value α. If l_i with the jth input fixed to $\overline{\alpha}$ is already in $\mathcal{S}$, it is removed from $\mathcal{S}$. In this way, the full controllability of each single instantiated LUT input is assured.

Since $k_i \leq 6$ for any $i \in \{1, \ldots, n\}$, the computational complexity of steps 6-20 is $O(2nk(t_1 + t_2))$, where t_1 is the time to upload $\mathcal{B}^*$ into the FPGA (step 10) and t_2 is the time required to upload and observe the output of $\mathcal{B}^*$ in order to check its equivalence with $\mathcal{B}$ (step 11). Although the worse case complexity of equivalence checking is exponential in the number of primary inputs of the design implemented by $\mathcal{B}$, we found that cryptographic algorithms are quite sensitive to changes. In our SNOW 3G case study, observing 20 output words (640 keystream bits) was enough to get a list that contained all obfuscated LUTs in the design.

At steps 21-30, we repeat the process for multiple stuck-at faults at instantiated inputs of each LUT in $\mathcal{S}$. First, the number of occurrences of each LUT l in $\mathcal{S}$, $N(l)$, in counted. Since 4-tuples representing the same LUT with different instantiated inputs appear in $\mathcal{S}$ in order, the counting can be performed in $O(|\mathrm{S}|)$ time by recording the number of LUTs with the same coordinate c in $\mathcal{S}$.

Let $\{\alpha_1, \ldots, \alpha_{N(l)}\}$ be a set of constants assigned to the inputs $\{j_1, \ldots, j_{N(l)}\}$ of a LUT l in $\mathcal{S}$. At steps 22-30, for each l in $\mathcal{S}$ and each subset J of the set $\{j_1, \ldots, j_{N(l)}\}$ of size greater than 1, the truth table of l in $\mathcal{B}$ is modified to a truth table in which all inputs in the subset J are stuck-at the corresponding constants in the subset A of the set $\{\alpha_1, \ldots, \alpha_{N(l)}\}$. As in the single stuck-at fault case, the modification is done directly in the bitstream.

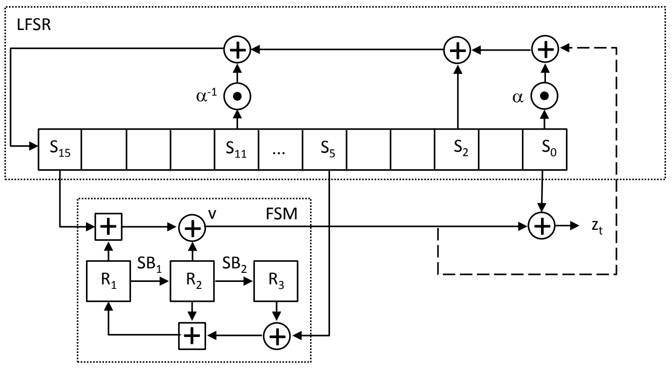

Fig. 2. SNOW 3G block diagram.

The resulting modified bitstream $\mathcal{B}^*$ is uploaded to the FPGA and executed to compare its output sequence to the one of $\mathcal{B}$. If the sequences are the same, then l is added to the set $\mathcal{M}$ along with its coordinate c, inputs J and multiple stuck-at fault values A.

Since $N(l) \leq 6$ for any l, the computational complexity of steps 22-30 is $O(n2^k(t_1 + t_2))$.

The algorithm terminates by returning the union of $\mathcal{S} \cup \mathcal{M}$.

V. CASE STUDY: SNOW 3G STREAM CIPHER

In this section, we demonstrate the feasibility of FINDOBFUSCATED() algorithm in the example of SNOW 3G stream cipher implemented in a Xilinx 7-series FPGA (XC7A35T-2CPG236). In the experiments, we used a VHDL implementation of SNOW 3G given to us by the authors of SNOW 3G.

A. *SNOW 3G design description*

SNOW 3G is the backbone of the 3GPP confidentiality and integrity algorithms UEA1 and UIA2. It is also used in 4G LTE and NG 5G standards. SNOW 3G is resistant to classical cryptanalysis [41]–[45], however physical attacks on its implementations through cache timing side-channels [46], electromagnetic inference analysis [47], transient fault injection [48], and bitstream modification [7] have been reported.

SNOW 3G is a word-oriented binary additive stream cipher [49] which takes as input a 128-bit *Initialization Vector* (IV) and a 128-bit secret key and produces a pseudorandom sequence called *keystream*. Each keystream element is a 32-bit word. The encryption/decryption is performed by combining the keystream with plaintext/ciphertext.

Fig. 2 shows a block diagram of SNOW 3G. The cipher consists of a 16-stage LFSR and a non-linear FSM. SNOW 3G has two modes of operation - initialization and keystream generation. In the initialization mode, marked by a dashed line in Fig. 2, the LFSR is loaded with a combination of the key and IV, the FSM is loaded with an all-0 state, and the cipher is clocked for 32 cycles without producing any output. After that, the cipher enters the keystream generation mode, marked by a solid line in Fig. 2, in which one keystream word is generated at each clock cycle.

B. Obfuscated SNOW 3G implementation

We created a protected implementation of SNOW 3G in which the part sensitive to fault injections - the FSM output - is obfuscated using an FSM-based hardware opaque predicate.

The Boolean expression realizing the SNOW 3G FSM's output is complemented with a redundant input, *opq* as $(S_{15} + R_1) \oplus (R_2 \odot \mathit{opq})$. The input *opq* comes from a state register of another FSM which always has a high value when the SNOW 3G FSM output is evaluated during the execution of the algorithm.

It was shown in [7] that a stuck-at-0 fault injected at the FSM output during the initialization can be exploited to extract the secret key. This is because, in this case, the LFSR state after the initialization depends entirely on the characteristic polynomial of the LFSR. Thus, by analysing the keystream, it is possible to reverse the LFSR to its initial state and recover the key-IV combination used to initialize it.

However, if the logic implementing the FSM output is obfuscated as we described above, the attack presented in [7] fails because it cannot find LUTs implementing the SNOW 3G's FSM output function in the bitstream (since SNOW 3G is word-oriented, the FSM output is 32-bit).

C. Deobfuscating SNOW 3G

We developed a software package implementing FINDOBFUSCATED() algorithm. The package uses the project Xray [17] to reverse engineer the bitstream format, python scripts to automate the processing of the PIP and LUT lists extracted from the bitstream, and *tcl* scripts for automating the upload of the bitstreams into the FPGA.

We used the package to deobfuscate the protected implementation of SNOW 3G described in the previous section. The size of the LUT list $\mathcal{L}$ recovered by reverse engineering at step 2 of FINDOBFUSCATED() was $n = 3,107$ (the number of LUTs reported by Vivado is $n = 3,053$). The size of the PIP list $\mathcal{P}$ recovered by reverse engineering at step 3 was $\sum_{i=1}^{n}(n \cdot k_i) = 12,533$ (compare to $6n = 18,642$). It takes 24 seconds to compute both lists.

The modified bitstreams $\mathcal{B}^*$ created at steps 5-8 were uploaded to the FPGA one by one at step 9. It takes $t_1 + t_2 =$ 6.3 sec on average to upload one bitstream into the FPGA, to generate 20 keystream words (640 bits) of $\mathcal{B}^*$ and to verify the equivalence of the keystreams of $\mathcal{B}^*$ and $\mathcal{B}$.

The set of deobfuscated LUT candidates returned FINDOBFUSCATED() contained all LUTs implementing SNOW 3G FSM output because redundant inputs of these LUTs behave as undetectable stuck-at faults during the execution of SNOW 3G. Since all points of interest for fault injection are discovered, after deobfuscation it becomes possible to extract the secret key of SNOW 3G through a bitstream modification attack [7].

Note that the hardware opaque predicate we considered is just an example. It's stealthiness was not considered because it can be replaced by any other, more stealthy, opaque predicate. This is because the hardware opaque predicate stealthiness does not affect the success rate of FINDOBFUSCATED() since it does not search for the hardware opaque predicate itself, but for the LUT inputs that behave as undetectable stuck-at faults during the execution of the algorithm.

VI. CONCLUSION

We presented a new FPGA design de-obfuscation method based on ensuring the full controllability of each instantiated LUT input via iterative LUT modifications at bitstream level. The shortcoming of the presented method is that it may not be able to deobfuscate constructions involving triple modular redundancy [13] or other methodologies that would mask the effect of the introduced faults. Thus, creating new low-overhead obfuscation techniques based on such concepts is worth investigating.

It is important to stress that real attackers are not concerned with the sophistication of the method they use. They can try anything, as long as it delivers results within a reasonable time. We implemented the presented algorithm in a software package and demonstrated its feasibility on the example of SNOW 3G stream cipher. Further work is required to evaluate it more thoroughly.

By providing a novel methodology for testing the resistance of obfuscation strategies, our findings are expected to contribute to the assurance of FPGA design security.

The presented method might not be able to deobfuscate constructions involving triple modular redundancy [13] or other types of functional duplication which mask the effect of faults. Thus, creating new low-overhead obfuscation techniques based on such constructions is interesting future work.

REFERENCES

[1] A. C. Aldaya, A. J. C. Sarmiento, and S. Sánchez-Solano, "AES T-Box tampering attack," *Journal of Cryptographic Engineering*, vol. 6, no. 1, pp. 31–48, [2015] 2016.

[2] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice—hardware trojan against a high-security USB flash drive," *Journal of Cryptographic Engineering*, vol. 7, pp. 199–211, Sep [2016] 2017.

[3] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (BiFI)–automated fault attacks against SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, 2017.

[4] S. Wallat, M. Fyrbiak, M. Schlögel, and C. Paar, "A look at the dark side of hardware reverse engineering-a case study," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pp. 95–100, IEEE, 2017.

[5] D. Ziener, J. Pirkl, and J. Teich, "Configuration tampering of BRAM-based AES implementations on FPGAs," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–7, IEEE, 2018.

[6] M. Ender, P. Swierczynski, S. Wallat, M. Wilhelm, P. M. Knopp, and C. Paar, "Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 112–119, 2019.

[7] M. Moraitis and E. Dubrova, “Bitstream modification attack on SNOW 3G,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1275–1278, IEEE, 2020.

[8] K. Ngo, E. Dubrova, and M. Moraitis, “Attacking Trivium at the bitstream level,” in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pp. 640–647, IEEE, 2020.

[9] M. Moraitis, E. Dubrova, and K. Ngo, “Breaking ACORN at bitstream level,” in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pp. 117–122, IEEE, 2020.

[10] D. Sisejkovic, F. Merchant, L. M. Reimann, H. Srivastava, A. Hallawa, and R. Leupers, “Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 3, pp. 1–26, 2021.

[11] R. Karam, T. Hoque, S. Ray, M. Tehranipoor, and S. Bhunia, “Robust bitstream protection in FPGA-based systems through low-overhead obfuscation,” in *2016 Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, IEEE, 2016.

[12] H. M. Kamali, K. Z. Azar, K. Gaj, H. Homayoun, and A. Sasan, “Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection,” in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 405–410, IEEE, 2018.

[13] T. Hoque, K. Yang, R. Karam, S. Tajik, D. Forte, M. Tehranipoor, and S. Bhunia, “Hidden in plaintext: an obfuscation-based countermeasure against FPGA bitstream tampering attacks,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 1, pp. 1–32, 2019.

[14] B. Olney and R. Karam, “Tunable FPGA bitstream obfuscation with boolean satisfiability attack countermeasure,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 2, pp. 1–22, 2020.

[15] V. Sergeichikand and A. Ivaniuk, “Implementation of opaque predicates for FPGA designs hardware obfuscation,” *Journal of Information, Control and Management Systems*, vol. 12, no. 2, 2014.

[16] M. Hoffmann and C. Paar, “Stealthy opaque predicates in hardware-obfuscating constant expressions at negligible overhead,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 277–297, 2018.

[17] SymbiFlow, “Project X-Ray.” https://prjxray.readthedocs.io/en/latest/, 2018.

[18] A. Chakraborty, N. G. Jayasankaran, Y. Liu, J. Rajendran, O. Sinanoglu, A. Srivastava, Y. Xie, M. Yasin, and M. Zuzak, “Keynote: A disquisition on logic locking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 1952–1972, 2019.

[19] M. Moraitis and E. Dubrova, “FPGA bitstream modification with interconnect in mind,” in *Proc. of the 9th Int. Workshop on Hardware & Architectural Support for Security & Privacy (HASP’20)*, ACM, 2020.

[20] A. D. Friedman, “Fault detection in redundant circuits,” *IEEE Transactions on electronic Computers*, no. 1, pp. 99–100, 1967.

[21] M. H. Schulz and E. Auth, “Improved deterministic test pattern generation with applications to redundancy identification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 811–816, 1989.

[22] M. Berkelaar and K. van Eijk, “Efficient and effective redundancy removal for million-gate circuits,” in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, p. 1088, IEEE, 2002.

[23] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, “Novel bypass attack and BDD-based tradeoff analysis against all known logic locking attacks,” in *International conference on cryptographic hardware and embedded systems*, pp. 189–210, Springer, 2017.

[24] Y. Shen and H. Zhou, “Double dip: Re-evaluating security of logic encryption algorithms,” in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pp. 179–184, 2017.

[25] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, “Removal attacks on logic locking and camouflaging techniques,” *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 517–532, 2017.

[26] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, “AppSAT: approximately deobfuscating integrated circuits,” in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 95–100, IEEE, 2017.

[27] Y. Shen, Y. Li, A. Rezaei, S. Kong, D. Dlott, and H. Zhou, “BeSAT: Behavioral SAT-based attack on cyclic logic encryption,” in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 657–662, 2019.

[28] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, “SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.

[29] M. John, A. Hoda, R. Chouksey, and C. Karfa, “SAT based partial attack on compound logic locking,” in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 1–6, IEEE, 2020.

[30] M. Harihara and P. Menon, “Identification of undetectable faults in combinational circuits,” in *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 290–291, IEEE Computer Society, 1989.

[31] P. R. Menon and H. Ahuja, “Redundancy removal and simplification of combinational circuits,” in *Digest of Papers. 1992 IEEE VLSI Test Symposium*, pp. 268–273, IEEE, 1992.

[32] M. A. Iyer and M. Abramovici, “FIRE: A fault-independent combinational redundancy identification algorithm,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 4, no. 2, pp. 295–301, 1996.

[33] M. Yasin, B. Mazumdar, J. J. Rajendran, and O. Sinanoglu, “SARLock: SAT attack resistant logic locking,” in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 236–241, IEEE, 2016.

[34] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan, “Provably secure camouflaging strategy for IC protection,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 38, no. 8, pp. 1399–1412, 2017.

[35] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, “Camoperturb: Secure ic camouflaging for minterm protection,” in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2016.

[36] S. Roshanisefat, H. Mardani Kamali, and A. Sasan, “SRCLock: SAT-resistant cyclic logic locking for protecting the hardware,” in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 153–158, 2018.

[37] J. Kim, J. P. M. Silva, H. Savoj, and K. A. Sakallah, “RID-GRASP: redundancy identification and removal using GRASP,” in *International Workshop on Logic Synthesis*, Citeseer, 1997.

[38] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[39] A. Kuehlmann and F. Krohm, “Equivalence checking using cuts and heaps,” in *Proceedings of the 34th annual Design Automation Conference*, pp. 263–268, 1997.

[40] M. Schmid, D. Ziener, and J. Teich, “Netlist-level IP protection by watermarking for LUT-based FPGAs,” in *2008 Int. Conf. on Field-Prog. Tech.*, pp. 209–216, 2008.

[41] A. Biryukov, D. Priemuth-Schmid, and B. Zhang, “Analysis of SNOW 3G resynchronization mechanism,” pp. 327–333, 01 2010.

[42] A. Kircanski and A. M. Youssef, “On the sliding property of SNOW 3G and SNOW 2.0,” *IET Information Security*, vol. 5, no. 4, p. 199, 2011.

[43] J. GUAN, L. DING, and S.-K. LIU, “Guess and Determine Attack on SNOW3G and ZUC [J],” *Journal of Software*, vol. 6, pp. 1324–1333, 2013.

[44] M. S. N. Nia *et al.*, “Improved Heuristic guess and determine attack on SNOW 3G stream cipher,” in *7’th Int. Symp. on Telecommunications (IST’2014)*, pp. 972–976, IEEE, 2014.

[45] A. Biryukov *et al.*, “Multiset collision attacks on reduced-round SNOW 3G and SNOW 3G$^{\oplus}$,” in *Int. Conf. on Applied Crypt. and Network Security*, pp. 139–153, Springer, 2010.

[46] B. B. Brumley, R. M. Hakala, K. Nyberg, and S. Sovio, “Consecutive S-box lookups: A Timing Attack on SNOW 3G,” in *Int. Conf. on Information and Communications Security*, pp. 171–185, Springer, 2010.

[47] J. Takahashi *et al.*, “Feasibility of fault analysis based on intentional electromagnetic interference,” in *2012 IEEE Int. Symp. on Electromagnetic Compatibility*, pp. 782–787, IEEE, 2012.

[48] B. Debraize *et al.*, “Fault analysis of the stream cipher SNOW 3G,” in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 103–110, IEEE, 2009.

[49] M. Robshaw, “Stream ciphers,” Tech. Rep. TR - 701, July 1994.